C/C++ Contributing Editors

The New C: It All Began with FORTRAN

Randy Meyers

Sometimes the best way to improve a language is to make it look more like the one it set out to obsolete 30 years earlier.

It all began with FORTRAN.

In this case, I am not referring to the fact that FORTRAN was the first programming language, but that a disagreement in the 1960s on how to implement parameter passing in FORTRAN accidentally gave FORTRAN a huge performance advantage on supercomputers in the 1970s, and eventually led to a new feature in C99 in the 1990s. The new C99 feature is restricted pointers, and one of the best ways to understand the motivation for restricted pointers is to follow the history of the accident in FORTRAN.

In FORTRAN, unlike C, if a function assigns a new value to a parameter, the value of the actual argument passed to the function will change, and after the function returns, the caller will see the new value of the argument. Consider the FORTRAN code in Example 1 below. If you call **F** passing a variable **Y** as an argument, after **F** returns, **Y** will equal 6.

Example 1:

```
SUBROUTINE F(X)
INTEGER X
X = 6
END
```

Here is where the disagreement came in. Different FORTRAN compilers chose one of two different ways to achieve the FORTRAN semantics for parameter passing. The first way was "by reference" parameter passing, or the way that your typical C programmer would write a function that modifies a variable in its caller. The function is passed the address of the argument and uses indirection everywhere when accessing the parameter. If the FORTRAN compiler produced C code as its output, the result would be something like Example 2.

Example 2:

```
void f(int *x)
   {
   *x = 6;
   }
```

However, on some machines, indirection was fairly expensive compared to directly referencing a local variable. This led to the second way to implement FORTRAN parameter passing. The address of the actual argument is still passed to the function,

but the function when entered makes a local copy of the argument, uses the local copy throughout the body of the function, and then copies the local copy back to the original argument before returning. Such a FORTRAN compiler would produce code that looks something like Example 3. While "copy in/copy out" parameter passing adds overhead to entering and returning from a function, if the parameter is referenced just a few times, and the (expensive on some machines) indirection is eliminated, the net result is increased performance (on some machines).

Example 3:

```
void f(int *x)
   {
   int xcopy = *x;
   xcopy = 6;
   *x = xcopy;
   }
```

Many times, the details of how a compiler implements a language feature are just "implementation details" in the usual sense. They do not affect the way programmers write programs, and programming language standards committees allow implementations to freely choose among the alternatives. However, FORTRAN programs could produce different results depending on the parameter passing mechanism used. Consider the FORTRAN code in Example 4, and the two different translations of that code into C.

Example 4:

```
SUBROUTINE G(X, Y)
INTEGER X, Y
X = 1 + X
Y = 5 + Y
END

// Translation using "by reference"
void g(int *x, int *y)
{
   *x = 1 + *x;
   *y = 5 + *y;
}

// Translation using
// "copy in/copy out"
void g(int *x, int *y)
{
   int xcopy = *x;
   int ycopy = *y;
   xcopy = 1 + xcopy;
   ycopy = 5 + ycopy;
   *x = xcopy;
   *y = ycopy;
}
```

The **G** subroutine adds different constants to both its parameters. If you pass two arguments, variables **A** and **B**, to the function **G**, and before the call **A** has the value

1 and **B** had the value 10, then as you would suspect, after the call, **A** would have the value 2 and **B** would have the value 15. This would be true regardless of which implementation of FORTRAN parameter passing was used.

However, consider what happens if you pass **A** (initial value 1) as both parameters. If "by reference" parameter passing is used, after the call **A** has the value of 7. **A** is updated during the assignment to ***x**, and then again during the assignment to ***y**, since both **x** and **y** point to **A**. In contrast, if "copy in/copy out" parameter passing is used, then after the call to **G**, **A** has the value 6. Distinct copies of **A** were made inside **G**, and the additions were made separately to each copy. Both copies were eventually assigned back to **A**, but the last assignment "won."

The two alternative parameter passing mechanisms are an example of the types of tradeoffs that the definers of any programming language have to face. Should the language require a particular implementation technique, possibly at the expense of efficiency? Should the language features be changed to avoid some controversy? The FORTRAN definition decided in favor of efficiency to allow either parameter passing mechanism. Once that decision was made, certain types of programs became non-portable, and were "outlawed."

The FORTRAN 66 standard contained all sorts of rules that seemed completely unmotivated to many programmers. You can pass any one variable to a function at most once in the argument list. If you pass a variable as an argument to a function, that function cannot also reference the variable as a global (FORTRAN **COMMON**). If you pass a variable to a function, you cannot also pass anything, and the function can not reference anything, that overlaps it in storage (FORTRAN **EQUIVALENCE**). No program that follows those rules can determine which parameter passing mechanism is used.

About ten years later, the Cray 1 supercomputer required highly optimizing compilers to allow traditional programs to make use of the machine's vector registers, and thus achieve supercomputer performance. Consider the program in Example 5. The most efficient code for the function is to load the array pointed to by **x** into a vector register, then load the array pointed to by **y** into a vector register, and then do a vector add instruction adding the two vector registers together. If the compiler generates vector instructions, rather than a traditional loop accessing the array an element at a time, the code may run an order of magnitude faster.

Example 5:

```
void
vector_add(float *x, float *y,
   float *result)
   {
   int i;
   for (i = 0; i < 64; ++i)
      result[i] = x[i] + y[i];
   }
```

The optimizer in a compiler can certainly transform a loop into a sequence of vector instructions, but the question is whether the sequence of vector instructions is really equivalent to the original loop. Loading all the **x** array and the **y** array into vector registers before doing any stores into the result array works only if you know that the result array is distinct from both **x** and **y**. Consider what happens if **result** points to **x[1]**. Then **result[0]** is really **x[1]**, and **result[i]** is the same as **x[i+1]**. Each iteration

of the loop stores values referenced in the next iteration. If you load **x** into a vector register before doing stores into result, the values calculated change. It is at this point the accident in the definition of FORTRAN comes in. In order to avoid requiring a particular parameter passing mechanism, the FORTRAN Standard had exactly the right set of rules to allow a vectorizing compiler to assume that **x**, **y**, and **result** are all distinct, non-overlapping arrays. Thus, by accident, FORTRAN had a major performance advantage on vector machines.

This performance advantage also carried over into parallel processor machines. The goal for an optimizing compiler for a parallel processor is to divide a loop into several ranges of iterations that can be done by separate processors working independently of each other. Thus, the loop in **vector_add** might be divided into two ranges: one processor might handle iterations 0 to 31 while another processor simultaneously does iterations 32 to 63. Work can be divided up among different processors only if the compiler knows that the results of the iterations done by one processor are not needed by a different processor working simultaneously. Proving this usually means the compiler needs to determine that different arrays are distinct, non-overlapping objects. Again, the rules in FORTRAN are exactly what a compiler needs in order to prove it can automatically parallelize a program.

While the FORTRAN performance advantage started out on supercomputers, it is now present on your typical, modern, general-purpose microprocessor. These days, most general-purpose microprocessors have a superscalar design:

- Instructions have separate stages of execution.
- A pipeline allows a small number of instructions to be in the process of execution, each instruction at a different stage in its execution.
- Multiple functional units may allow separate operations to overlap, such as an add with a multiply.
- The processor runs at a multiple of the memory speed, and typically runs faster than the cache.
- Even the registers might not be fast enough. If one instruction stores a value in a register, and the next instruction tries to use that register as an operand, then the machine may have to stall waiting for that result to become available.

A compiler for a superscalar machine has to carefully schedule the instructions it generates, and interleave instructions from different computations, to achieve maximum performance. For example, loading a memory value into a register takes a long time. If you attempt to use the value in the register before the value from memory arrives, the processor will stall. So, the compiler will try to generate a load instruction, followed by several instructions that do not need the result of the load, followed by the instructions that use the result of the load. This gives the load time to finish, and keeps the processor busy.

Consider **vector_add** from Example 5. A typical optimization performed for a superscalar processor is to load **x[i+1]** and **y[i+1]** into registers at the start of the iteration for **i**. This means that each iteration of the loop conveniently has its operands fetched from memory for it by the previous iteration, and thus never has to wait for the slow memory system to provide a value.

However, this prefetching works only if the results of one iteration of the loop does not affect the operands fetched by the next iteration of the loop. Proving this usually means knowing that the objects operated upon by the loop are distinct and non-overlapping.

Clearly, C could not concede this performance advantage to FORTRAN forever. Note that FORTRAN's advantage over C comes only because C programs use pointers. A C compiler can perform any of the optimizations discussed when it sees individual objects being operated upon directly. The problem is when objects are operated upon indirectly through pointers. Since, in general, the compiler has no idea which object a pointer is referencing, the compiler cannot prove a pointer references a distinct object.

One approach used by some C compilers is to provide an optional, "unsafe" level of optimization beyond normal optimization. If unsafe optimization is enabled when the program is compiled, the compiler will assume that all pointers point to distinct objects, and the resulting program will run faster. Unfortunately, if the programmer meant for some pointers to reference the same object, the program will likely also get incorrect results.

A better solution is to allow the programmer to mark which pointers in the program point to distinct objects. This is the new "restricted pointer" feature in C99. (Some C++ compilers also support restricted pointers; however, they are not part of Standard C++.) C99 has a new keyword, **restrict**, that is a type-qualifier like **const** and **volatile**. Syntactically, **restrict** can appear wherever **const** and **volatile** can. However, semantically, **restrict** must modify a pointer type. This usually means that **restrict** follows the ***** in a pointer declaration. The following snippet shows some examples.

Example 6:

```
int *restrict p1;    // OK, p1 is a restricted pointer to int
restrict int *p2;    // invalid: pointer to restricted int
typedef int *intptr;
restrict intptr p3;  //OK, p3 is a restricted pointer to int
```

C99 also adds a new syntactic spot where any type qualifier can appear: after the **[** in an array parameter declaration. Remember, a parameter of type array is really a parameter of type pointer. The type qualifiers following the **[** act as if they followed the ***** when the parameter's type is rewritten as a pointer type. Thus:

```
int f(float array[const restrict]);
```

is the same as:

```
int f(float *const restrict array);
```

There's a bug in the C99 Standard — the change to the grammar to allow type qualifiers following the **[** in an array parameter declaration was made only to the grammar rule when the parameter is named. The change should have also been made for unnamed parameters. I hope that C compilers will accept the unnamed parameter case as a common extension until the C Standard can be corrected:

```
int f(float [restrict]);  // common extension
```

As you probably suspect, the rough meaning of a restricted pointer is that it provides a unique way to access a unique, non-overlapping object in the program. In other words, the object accessed through this pointer can be assumed to be unique from any other object being referenced in the code during the lifetime of the restricted

pointer, or until the restricted pointer's value changes. This allows an optimizer to safely make the sorts of optimizations discussed above, as well as other optimizations such as avoiding recomputing common (duplicate) subexpressions involving access through pointers.

There is an exception to the rule that a restricted pointer during its lifetime provides the sole access to an object. If the object is never modified in any fashion during the lifetime of the restricted pointer, then the object may be accessed not only by the restricted pointer but other ways as well. Such access does not interfere with any optimizations.

In order to understand the "uniqueness" rules of restricted pointers, you might want to think of them in terms of "copy in/copy out" semantics. Pretend that when you assign an address to a restricted pointer, a copy is made of the entire part of the object that is being referenced though the restricted pointer. The restricted pointer then points to the copy instead of the original object. Magically, the restricted pointer remembers the address of the original object, and if you ever change the value of the restricted pointer, or the restricted pointer goes out of scope, then the copy of the object is copied back to the original object referenced. This pretend "copy in/copy out" transformation breaks your program only if the program violates the uniqueness rules of restricted pointers.

Here are some examples. If someone stores into the original object during the lifetime of the restricted pointer, then accesses through the restricted pointer will fetch the unmodified values from the copy, not the updated values from the original object. If the restricted pointer is used to modify the object, then accesses made not through the restricted pointer during the lifetime of the restricted pointer will see the old values from the original object, not the updated values in the copy. On the other hand, if your program works even if the pretend copy in/copy out transformation was performed, then it is safe to use restricted pointers, and doing so might enable many useful optimizations.

The lifetime of a restricted pointer is the period of time during the execution of the program that the pointer itself would be allocated. For an automatic variable, it is the block in which it is declared is active. For a parameter, it is until the function body returns. For a file-scope variable, it is until **main** returns.

As I stated above, the uniqueness property applies only to the parts of the object actually accessed through the restricted pointer. For example, if **vector_add** from Example 5 declared all three of its parameters to be restricted pointers, it would be valid to make the call:

```
float data[192];
vector_add(&data[0], &data[64], &data[128]);
```

Since **vector_add** accesses only 64 elements from any of the arrays pointed to by its parameters, all of the accesses of the data array are unique to one of the three restricted pointer parameters. The data array is really being treated as three, unique, non-overlapping arrays. However, if the second argument was **&array[63]**, the call would be invalid since the **x** parameter and the **y** parameter were both accessing the same storage.

It is best to confine your use of restricted pointers to code that is fairly straightforward. If you have a hard time following how your pointers are used, there is an increased danger that you are inadvertently making invalid references to the

object referenced by the restricted pointer not using the restricted pointer (you have invalid aliasing).

This brings us to the subject of debugging. If your program doesn't work, or stops working in the future, and you use restricted pointers, you might want to see if the problem is invalid aliasing. Some compilers have an option to merely ignore **restrict**, which allows you to test if aliasing is the problem. If your compiler lacks such an option, define **restrict** to be a macro with no body when compiling your program. Since the only meaning of **restrict** is to permit additional optimization, it can be safely deleted from any program.

In addition to optimization, you can also use **restrict** to warn a user of your code that the code just will not work if you pass in pointers to the same or overlapping objects. For example, the C89 standard contained a statement at the beginning of the library section that, unless stated otherwise, you must not pass pointers to the same or overlapping objects to the functions in the standard library. This ruled out lots of silly uses of the library, like calling **sprintf** with the pointers to the format string and the result string being the same. The C99 Standard still contains that prose prohibition, but emphasizes the requirement by making most of the pointer arguments in the prototypes in the library restricted pointers. The **restrict** keyword allows the specification in the C Language itself a prohibition that previously required English text.

There you have it. Thirty-three years of programming language history and a new keyword in C.

Randy Meyers *is consultant providing training and mentoring in C, C++, and Java. He is the current chair of J11, the ANSI C committee, and previously was a member of J16 (ANSI C++) and the ISO Java Study Group. He worked on compilers for Digital Equipment Corporation for 16 years and was Project Architect for DEC C and C++. He can be reached at* ***rmeyers@ix.netcom.com****.*